Implementation of REST API vs GraphQL in Microservice Architecture

Naman Vohra
Computer Science Department,
Faculty of Computing and Media
Bina Nusantara University
Jakarta, Indonesia 11480
naman.vohra@binus.ac.id

Ida Bagus Kerthyayana Manuaba
Computer Science Department,
Faculty of Computing and Media
Bina Nusantara University
Jakarta, Indonesia 11480
imanuaba@binus.edu

2022 International Conference on Information Management and Technology (ICIMTech) | 978-1-6654-5090-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICIMTECH55957.2022.9915098

***Abstract*—Recently, microservices have become a popular web architecture that implements Application Programming Interfaces (API) for main communication between its services. The Representational State Transfer (REST) has been recognized as a popular framework to develop web APIs. However, the presence of "GraphQL," an alternative technology to REST, leads to a lot of debates and discussions to see which technology is more effective for communication between clients and server. Although many studies have discussed these two technologies, there is still limited study which covers the details on their implementation and performance in a microservice architecture scenario. Therefore, this paper discusses the implementation of REST and GraphQL technologies and performs a comparative study on the performance of each technology in a microservice architecture system, based on an identical freelance marketplace scenario. In addition, two corresponding gateways, the Ocelot and the HotChocolate, were implemented in this study. The gateways were tested in a local system on 17th March 2022. The test was conducted using a tool called "JMeter" which enables three test cases with distinct numbers of users and loop counts to be produced, with each test case impacting the gateways differently. For comparative analysis, the average "response time" and "throughput" parameters were analyzed. Results indicate that in every test case, REST and GraphQL perform comparably when dealing with queries of type POST and PUT but shows a significant performance difference when processing (GET) queries that involve data retrieval from more than one service.**

Keywords—Microservice, REST, GraphQL, Performance, Comparative Study

I. INTRODUCTION

Microservices refer to a set of small, independent services that work together [1]. Each service has its own specific functionality, and the ideal practice is to make the services loosely coupled [2], meaning minimizing the dependencies between the services. However, oftentimes, clients need to retrieve interrelated data coming from various services in a microservice system.

This task is no longer simple if database-per-service pattern is implemented as each service has its own database [3], unlike in a system with monolithic architecture where only a single database is used for data storage. This means that some queries will span multiple services.

One of the solutions to this problem is for the system to adopt the API composition pattern, that includes an API composer with the task of implementing a query that fetches data from multiple services by invoking each service through its API [3]. This is usually done by the API gateway.

As the name suggests, an API gateway acts as an access point (between the client and the services) that directs client requests to the appropriate microservices and combines the responses [4]. One of the goals of an API gateway is to enhance the performance of a system by reducing the number of requests per client [4].

Currently, there are two popular technologies that are commonly used to develop web APIs, they are known as the Representational State Transfer (REST) and GraphQL. There is a number of recent studies discussing these two technologies in web application development.

Study by Lawi et al., [5] successfully assessed the performance of REST and GraphQL in two separate systems in monolithic architectures application, which resulted in REST being 51% faster than GraphQL. However, in this study, GraphQL was proved to be more efficient in resource utilization.

Another study by Hartina et al., [6] also showed similar results and the system in which the performance of REST and GraphQL technologies was assessed comprised of an RDMS, a Laravel based web application, and a Nginx server, thus making the software architecture a monolithic one.

In addition, Diyasa et al., [7] conducted the experiment by testing two applications that access the same database. Results from this research indicate that REST has a better response speed than GraphQL, while GraphQL is superior in terms of bandwidth optimization.

Based on these related studies, there is none of them discussing the performance of GraphQL and REST in a microservice architecture application. There are only a few studies that have discussed the implementation of these individual technologies in microservices architecture scenario, however there are no comparison studies discussed further.

Hence, this study discusses the implementation and at the same time conducts a comparison between REST and GraphQL. For the purposes of the study, a set of two identical systems comprising of the same services but with different types of gateways was created. The first system contains a series of REST backend services with a REST compatible gateway while the second system is comprised of GraphQL APIs sported by a GraphQL gateway.

Both the gateways implement different mechanisms when acquiring data from the services, and therefore, the performance of GraphQL and REST can be evaluated by testing how well each type of gateway caters to the clients' requests and collaborate with the underlying services.

978-1-6654-5090-4/22/$31.00 ©2022 IEEE

In addition, the complexity needed in each gateway to provide the same results to the client will be investigated as well. In analysing the performance, both technologies are implemented through a system with microservice architecture that follows the database-per-service pattern.

Therefore, the objectives of this study are as follows:

- To evaluate REST and GraphQL in microservices architecture implementation
- To compare the performance of REST and GraphQL in a microservice system that is centred on the same scenario

In the next sections, the methods and the implementation of the gateways will be discussed further, followed by the results of the testing and the discussion. This study is closed by the conclusion and the suggested future works.

II. METHODS

A. Evaluated System Architecture

In this study, two identical microservice applications with different gateways had been developed. First application is implementing the *Ocelot* gateway to represent the REST technology, and the *HotChocolate* gateway in the other application for the GraphQL technology.

Please refer to Fig. 1 and Fig. 2 below to see the difference between these two implementations.

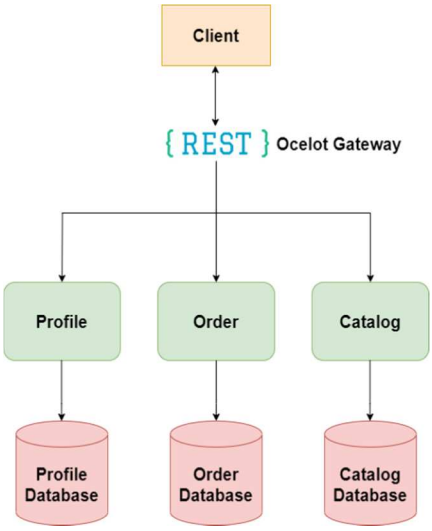

Fig. 1. Microservice system with REST gateway and services.

Fig. 1 depicts a microservice system comprising of RESTful services sported by Ocelot, a REST compatible gateway. The services are profile, order, and catalog. The system implements the API composition pattern, which was adopted to overcome the challenge of fetching data from multiple services with segregated databases.

In this system, Ocelot is the composer that manages the client request and returns the desired output to the client. The back-end framework used to develop the services is ASP.NET Core, while the RDMS utilized for data storage was Microsoft SQL server.

On the other hand, Fig. 2 illustrates the architecture of a microservice system comprising of a HotChocolate GraphQL gateway and three GraphQL APIs. However, the HotChocolate gateway has a unique feature called schema stitching that allows the merging of the services' schemas, which in turn were used to create a unified and customized schema at the gateway that is accessible to the client, as shown in Fig. 2. Other aspects of the system such as the architectural pattern, types of services, back-end framework, and the RDMS used remain the same.

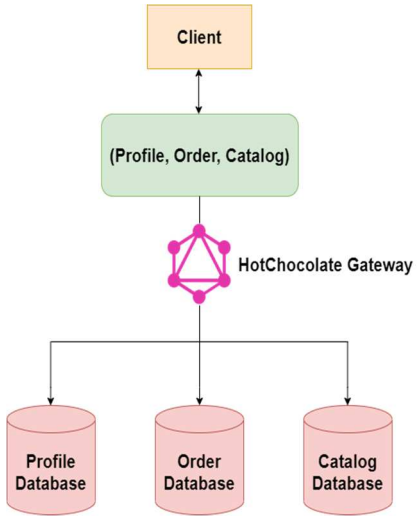

Fig. 2. Microservice system with GraphQL gateway and services.

The implementation section will cover the details of how each gateway was used as a composer, discussing the specialties of both the gateways, which are request aggregation and schema stitching respectively.

As for the services, the scenario that was used while designing them for both the systems was that of a simple online freelance marketplace. The catalog service manages all the job postings created by buyers, whose details are stored in the profile service. The profile service also contains information of another main type of user, which is the freelancer. The last service is the order service, that records the job postings (catalogs) and the corresponding freelancers that worked on these, including the payment and review information associated to each order. These services are adequate to generate simple POST to complex GET queries for testing the systems, with the details of the test queries discussed in the next subsection.

B. Test Scenarios

For comparison purposes, five different test queries had been set to test the performance of each technology, in a microservice architecture scenario. The details of the test queries can be seen in Table I below.

TABLE I. QUERY CANDIDATES FOR TESTING

Test Queries	Description	Services Spanned
Q1 (POST)	Simple login of a user	Profile
Q2 (PUT)	Update buyer's profile	Profile
Q3 (GET)	View information of all freelancers including the skills that they possess	Profile
Q4 (GET)	Display all buyer details and the corresponding job postings that they created	Profile Catalog
Q5 (GET)	Retrieve all the order details, including the freelancers and the catalogs' information associated to each order	Profile Catalog Order

978-1-6654-5090-4/22/$31.00 ©2022 IEEE

Table I illustrates the list of queries that were used for testing the REST and the GraphQL gateways. The first query simulates a simple log in process that returns a success message if the user exists in the system, while the second query aims to update information of an existing buyer profile.

The remaining queries are of the type GET as the focus is to assess how each type of gateway retrieves and aggregates data from varying number of services. The first GET query, which is Q3, fetches data from a single service. This test query represents a scenario where buyers visit the freelance marketplace website to look for potential freelancers and view their qualifications. Likewise, freelancers would be interested to see the list of the potential buyers (clients) and their job postings, with this situation depicted by the fourth query (Q4), which needs information from two services.

The last query (Q5) is the most complex query as it retrieves the orders' details with each order requiring information from the catalog and the freelancer services, to display the completed job posting and the associated freelancer that worked on it. Furthermore, every order detail also carries the associated payment and the review information. Just like the third and the fourth queries and for the purpose of testing, all the order details were fetched to analyze how the gateway performs when complexity in terms of the number of services spanned by the query and hence data count were increased.

C. Testing Scenario (Test Case)

In this study, three different test cases were designed to evaluate and compare the performance of the technologies with increasing user load. Detail of the test case can be seen in TABLE II below.

TABLE II. TEST PROPERTIES OF EACH TEST CASE AND DATA COUNTER PER SERVICE

Properties	**Values**
Number of threads (users)	Cases:- 1: 100 – 2: 250 – 3: 500
Ramp-up period (seconds)	5
Loop count	Cases:- 1: 5 – 2: 2 – 3: 1
Data Count per Service	
Service	***Data Count***
Profile	315
Catalog	1000
Order	1509

Note that although the total number of requests or samples in every test case will be the same, each test case will be leaving a different impact on the system performance. For instance, in the first case, each user will perform a test query 5 times (as the loop count is 5), however the remaining queries will start only after the first query or request is completed. But, in the third case, requests may hit the gateway in parallel, meaning that in the second and the third cases, the chances of the gateways receiving concurrent user requests are higher than the first case, thus generating more stress on the system.

The tool that was used for the performance testing of the systems was Apache JMeter (version 5.4.3). The number of threads signifies the number of users that will interact with the system. Ramp-up period refers to the time taken for all the users to be active, which means that in the first case with 100 users and 5 ramp up period, after every 0.05 seconds a new user will be created. The last property is loop count which indicates the number of times each user will create a request, which by default is 1.

Thus, JMeter allows a real case simulation of the test scenarios described in the previous subsection to be created. For example, in the third test query (Q3), when a buyer wants to view the list of potential freelancers, the gateway will first need to retrieve freelancers' information from the profile service and present it to the buyer. However, there will be hundreds of buyers accessing the page showing freelancers' details. Similarly, in the fourth test query, multiple freelancers will visit the system to look for clients and the created job postings. Therefore, JMeter will produce the stated number of users, with each user making the specified request (the test query) to the gateway.

D. Testing Flow

The REST and the GraphQL systems were both tested in a local system, which is ASUS VivoBook S14 S410U (upgraded RAM of 16 GB), on 17th March 2022. Further detail about the testing flow is demonstrated in Fig. 3 below.

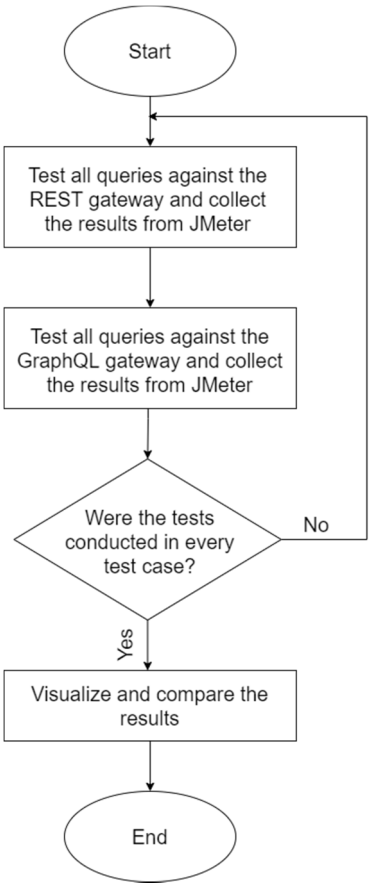

Fig. 3. Flowchart of the testing scenario.

Based on the flowchart diagram, firstly, the REST gateway was tested with each test query one at a time (e.g., with the Q1 scenario followed by Q2 and so on) under the condition of the first test case with 100 users and 5 loop count. The microservice application with the REST services was restarted after the testing of each query against it with JMeter, to ensure that the local system in which the application was running was under the optimal condition before the testing of the next query.

Followed by this, the GraphQL based microservice application was evaluated with the same set of queries and tested in a similar fashion as in the case of the REST based system. After both the gateways were tested in test case 1, the process is repeated with the next test case, which is test case 2 (250 users and loop count of 2), as illustrated in Fig. 3. Lastly, once the gateways were tested in all the three test cases, the results in the form of response time and throughput which indicate the performance of the REST and the GraphQL systems were then visualized and discussed in the results section.

III. Implementation

In this section, the fourth test query, which aims to retrieve all the buyers and the corresponding job postings posted by each buyer will be used as an example to demonstrate the implementation of each gateway's unique feature.

A. Ocelot Gateway

Ocelot is a gateway that is compatible with services that communicate using the HTTP protocol, making it work well with RESTful APIs. Fig. 4 demonstrates the key functionality of Ocelot, which is request aggregation.

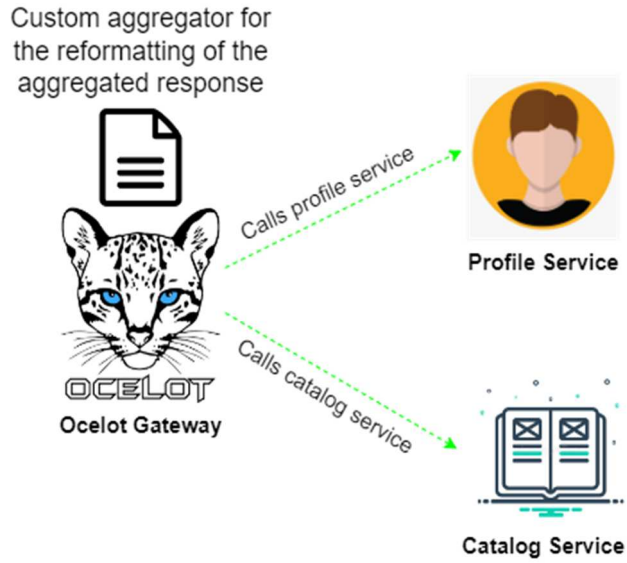

Fig. 4. Request aggregation.

Request aggregation allows the mapping of responses from multiple APIs into one object. In the case of the fourth scenario (Q4), the Ocelot gateway will invoke both the profile and the catalog services and merge their responses.

At this point, the merged result only contains two separate lists, each comprising of all the buyer records and the catalog records respectively, thus not reflecting the desired outcome of a single list of buyer records with each record having nested information of the catalogs posted by the buyer.

Therefore, a custom aggregator was created so that the merged response can be modified first before being returned to the client [8]. This file allows additional logic for querying and reformatting the combined response to be defined and then applied so that the client can receive the suitable output.

B. HotChocolate Gateway

HotChocolate refers to a .NET GraphQL platform that allows a GraphQL layer to be added to a new system or even an existing one [9]. What differentiates the HotChocolate gateway from other GraphQL services in the system is that it is not only a GraphQL server, but also a stitching server. Stitching is associated to the term schema stitching, a powerful feature that allows the GraphQL gateway to have access to the objects of the underlying services.

The objects can be shared between and used by the microservices. This powerful functionality of schema stitching is referred to as the delegation of resolvers [10]. Using Q4 as an example, the Id of the buyer object from the profile service was delegated to the catalogsByBuyer function in the catalog service.

The client will now be able to request all the buyer records with each of them having the corresponding list of catalogs (catalogsByBuyer) created by the buyer, as illustrated in Fig. 5. Moreover, no further reformatting of response needs to be done. It can be observed that the complexity needed to achieve the desired format is lower compared to that of the Ocelot gateway.

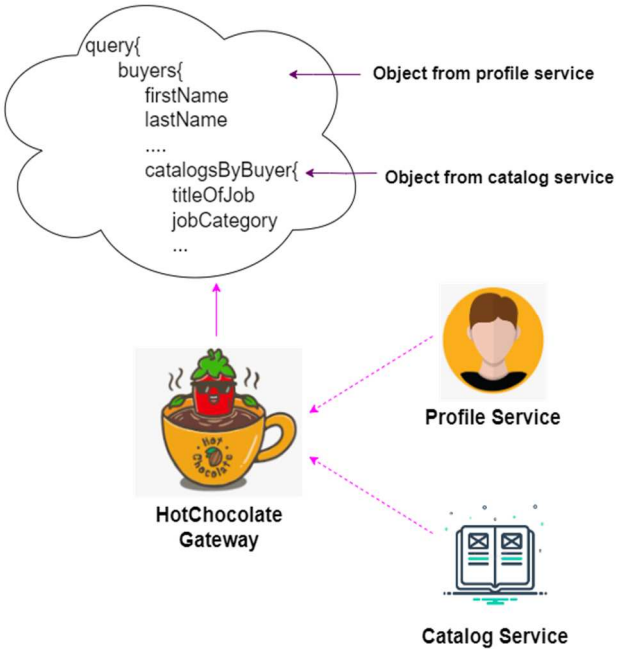

Fig. 5. Schema stiching functionality.

IV. Results

A. Response Time

Response time refers to the time taken for an application to process and respond to a client's request [11]. However, since large number of users and therefore requests were simulated in every test case, the average response time parameter was used instead to give a better overview of the performance of the REST and the GraphQL systems when servicing the client's queries. Fig. 6 illustrates the difference between the average response times of the GraphQL and the REST technologies for all scenarios in different test cases.

In all the test cases, the mean response time of the GraphQL and the REST gateways for the first two queries, which are of the method POST and PUT (also POST in the case of GraphQL) remain comparable.

978-1-6654-5090-4/22/$31.00 ©2022 IEEE

On the other hand, a significant difference can be observed between the response time of both the technologies when processing queries of type GET, particularly the ones that retrieve data from two services, which are Q4, and Q5. This trend applies to all the test cases except for the third query in the first case. It can be observed that in this situation, GraphQL was able to return all the freelancers' information with a time of 5.2 seconds on average, which is 0.2 seconds ahead of REST. For the remaining test cases, REST responded more rapidly than GraphQL. For example, in the first test case for the fourth and the fifth queries, REST was 2.25- and 5.8-times swifter than GraphQL, while in the second test case, the values are 1.6- and 5.8- times respectively. For the third test case, the ratios remain similar, which correspond to 2.5 and 4.3.

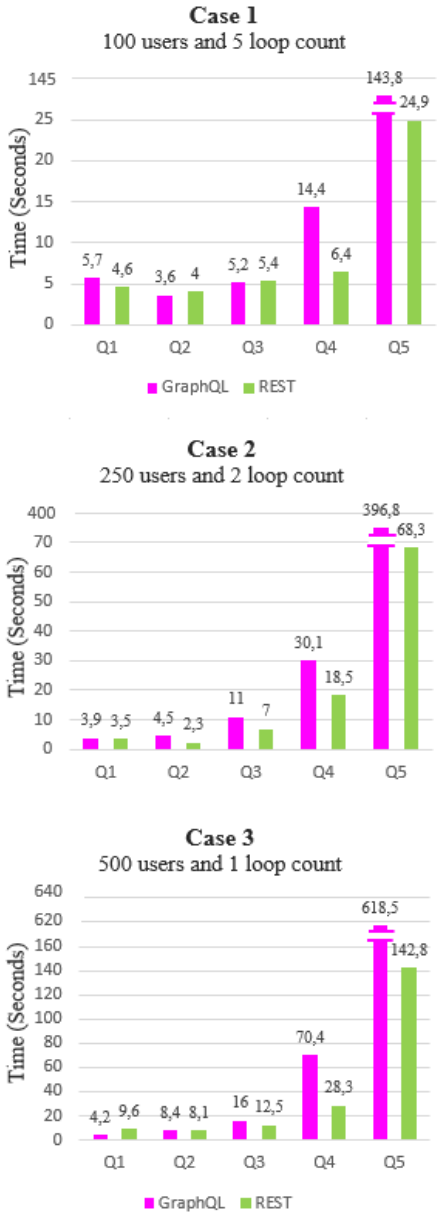

Fig. 6. Comparison between the average response times of REST and GraphQL for all queries in every test case.

B. *Throughput*

Throughput signifies the number of requests that is processed successfully by a system per unit time. Throughput is calculated by dividing the number of requests by the period from the start of the first sample to the end of the last sample. Fig. 7 provides an overview of the throughput values of the GraphQL and the REST gateways.

In all the test cases, it can be observed that the REST gateway could handle more client requests per second when compared to the GraphQL gateway, with some exceptions, as illustrated in certain scenarios in case 1 and case 3.

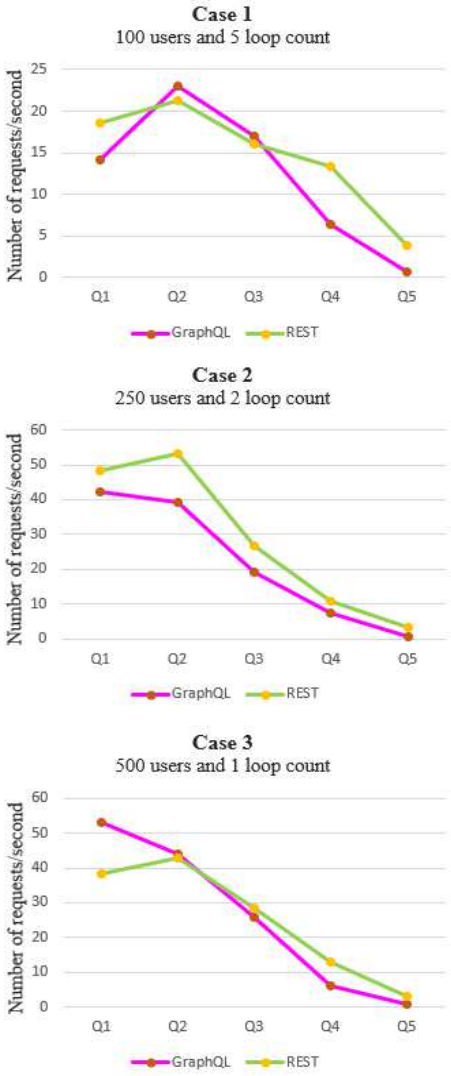

Fig. 7. Comparison between the throughput values of REST and GraphQL for all queries in every test case.

However, the performance of the gateways when running the fifth query in case 2 and case 3 provides an interesting finding which shows that the GraphQL gateway could no longer handle 250 users and 500 users making 500 requests with each of it fetching all the order details (Q5), as the throughput value for this query is near to 0.

For the REST gateway, it can still handle the load of processing Q5 with 0% error in case 2. However, in case 3, this is no longer the case. Another important finding is that the throughput values of the GraphQL gateway for the fifth query in all test cases explain the overwhelmingly large response time of this gateway when managing Q5 requests, as seen in Fig. 6.

978-1-6654-5090-4/22/$31.00 ©2022 IEEE
11-12 August 2022
2022 International Conference on Information Management and Technology (ICIMTech)

V. Discussion

It can be observed that the REST compatible Ocelot gateway performs significantly better than the GraphQL gateway especially when aggregating data from multiple services. Initially, in case 1 scenario 3 (Fig. 6), the GraphQL gateway was able to perform slightly better than the REST gateway when fetching freelancers' information. However, as the user load and the number of services encompassed by the query continue to increase, a huge difference in response time and hence the throughput can be observed.

Analysis shows that the huge gap in performance is due to the mechanisms of the gateways. For example, in scenario 4 where data is retrieved from two services, namely the profile and the catalog service, it takes the Ocelot gateway only two function calls to retrieve all the buyers and the catalogs from the profile and the catalog services respectively.

In the case of the HotChocolate gateway with the schema stitching functionality, the gateway hits the profile service to get all the buyer records and passes the Id of each buyer record to a function in the catalog service, that returns the list of catalogs for the corresponding buyer. The gateway will fetch the data once the lists of catalogs for all buyers are resolved in the catalog service [12] and then internally stitch these to the buyer records. It can be concluded that the cost of function calls in the catalog service and the internal stitching in the HotChocolate gateway (on the system performance) are greater compared to the manual mapping of each buyer record to the corresponding catalogs list in the Ocelot gateway.

Although present studies of REST and GraphQL in microservices architecture scenario are mostly focused on their implementation, a study that compares their performance was found, with GraphQL utilizing the graphql-dotnet library. Results indicate that GraphQL performed considerably worse than REST in all test cases that fetch different quantities of data [13].

However, it should also be noted that the code complexity in the REST gateway to achieve the desired response is greater when compared to the GraphQL gateway as a custom aggregator must be defined to reformat the aggregated response from the services. In addition, a new endpoint would need to be registered in the Ocelot gateway if there is a need to return a new set of data [13]. This is not the case with GraphQL, allowing for easier application development [14].

VI. Conclusion

In this study, two microservice systems with different technologies, which are the REST and the GraphQL were successfully developed. The REST compatible Ocelot gateway with the feature of request aggregation was used to communicate with the underlying REST services and return the client request, while for the GraphQL based system, the HotChocolate with the schema stitching functionality was utilized.

Results show that the REST gateway is superior when aggregating responses in various scenarios, resulting in faster response times and better throughput values when compared to GraphQL. However, unlike GraphQL, more effort is required in its implementation.

It is to be noted that this study poses a limitation as both gateways were tested in a local environment, which signifies that the results were dependent on the hardware specifications and the condition of the local machine at the time of testing.

For future works, alternative frameworks to HotChocolate, such as Apollo, Hasura, etc., that implement schema stitching can also be explored as these frameworks may possess specific features for optimizing the performance of GraphQL. The next major release of HotChocolate (version 13) that emphasizes on refining schema stitching can also be utilized [15]. Furthermore, pagination and caching can be used to further speed up GraphQL's response time. Lastly, this study can be enhanced by hosting each microservice system on a dedicated server to provide a more accurate and stable values for the response time and therefore the throughput.

References

[1] S. Newman, *Building Microservices: Designing Fine Grained Systems*. Sebastopol:O'Reilly, Inc., 2014.

[2] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st ed. Sebastopol: O'Reilly Media, Inc., 2016, pp 67.

[3] C. Richardson, *Microservices Patterns*. Shelter Island: Manning Publications CO., 2019, pp 221-223.

[4] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," in *CLOSER 2018 - 8th International Conference on Cloud Computing and Services Science*, pp. 221–232.

[5] A. Lawi, B. L. E. Panggabean, and T. Yoshida, "Evaluating Graphql and REST API Services Performance in a Massive and Intensive accessible information system," MDPI, vol. 10, no. 11, Nov. 2021.

[6] D. A. Hartina, A. Lawi, and B. L. E. Panggabean, "Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University," in *The 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT) 2018*, pp. 237–240.

[7] G. S. M. Diyasa, G. S. Budiwitjaksono, H. A. Ma'rufi, and I. A. W. Sampurno, "Comparative Analysis of Rest and GraphQL Technology on Nodejs-Based Api Development," in *5th International Seminar of Research Month 2020*, Apr. 2020, vol. 2021, pp. 1–9.

[8] T. Pallister, T. Loureiro, and Edwin, *Request aggregation*. [Online]. Available:https://ocelot.readthedocs.io/en/latest/features/requestaggregation.html. [Accessed: 23-Feb-2022].

[9] R. Staib, *Introduction*, 02-Dec-2020. [Online]. Available: https://chillicream.com/docs/hotchocolate/v10. [Accessed: 23-Feb-2022].

[10] T. Tengler, *Schema configuration*, 10-Aug-2021. [Online]. Available: https://chillicream.com/docs/hotchocolate/distributed-schema/schema-configuration. [Accessed: 23-Feb-2022].

[11] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "Performance Comparison between Container-based and VM-based Services," in *20th Conference on Innovations in Clouds, Internet and Network (ICIN),* Mar. 2017, pp. 185-190.

[12] T. Tengler, *Schema stitching*, 14-Jun-2021. [Online]. Available: https://chillicream.com/docs/hotchocolate/v10/stitching. [Accessed: 23-Feb-2022].

[13] P. Johansson, "Efficient Communication With Microservices," M.S. thesis, Umeå University, Umeå, 2017.

[14] A. Ritsilä, "GraphQL: The API Design Revolution," B.S. thesis, Haaga-Helia University of Applied Sciences, Helsinki, 2017.

[15] ChilliCream, *HC - 13.0.0* . [Online]. Available: https://github.com/ChilliCream/hotchocolate/milestone/65. [Accessed: 27-May-2022].